Spack-Based Packaging and Development for HEP

Chris Green[1,*], *James* Amundson[1], *Lynn* Garren[1], *Patrick* Gartung[1], and *Marc* Paterno[1]

[1]Scientific Computing Division, Fermi National Accelerator Laboratory

Abstract. The *art* event-based analysis framework is used by multiple Intensity Frontier and other experiments and projects in High Energy Physics (HEP). A system of tools and scripts based around a Fermilab-originated package management system called UNIX Product Support (UPS) has evolved to provide development, packaging, release management, and distribution of the *art* suite, related experimental software and external dependencies in relocatable binary form.

The existing system has limitations in environments such as macOS with System Integrity Protection (SIP) enabled and the various High Performance Computing (HPC) systems that the field of HEP is increasingly looking to leverage. This has led to a search for other ways of providing experiments with a packaging and build system that meets their needs. We describe our efforts to develop a packaging and release management system based on the Spack HPC package management tool to replace the existing UPS-based system. We additionally describe a companion system: SpackDev and cetmodules, allowing simultaneous development of multiple HEP software packages in a consistent environment. A successful implementation of the intended system would have applicability across HEP rather than merely to the users of the *art* framework.

1 Introduction

The *art*[1, 2] event-based analysis framework is a multi-package suite with ~ 15 external dependencies in addition to libraries from the operating system. It is used by multiple HEP experiments and projects[1][3] and supported with an effort equivalent to less than 3 fulltime people including development, build, test, release, distribution and support activities. The same team also produces release distributions for the software produced by these entities and the more than 120 external products upon which they rely. This is sustainable in part due to a development, packaging, build, release and distribution system that allows the creation of a binary distribution for a release consisting of packages of specified versions built with a particular compiler and language standard and with specified options for each package where appropriate. We describe this system and some of its essential characteristics, and motivations for replacing it. Additionally, we describe our efforts to implement a replacement system based around the Spack[4, 5] packaging system, and our plans for completion and deployment.

*e-mail: greenc@fnal.gov

[1]ArgoNeuT, DUNE, ICARUS, LArIAT, MicroBooNE, Mu2e, Muon g-2, NOνA, SBND, artdaq, and LArSoft

2 The Existing System and Motivation for Change

UPS[6, 7] was developed at Fermilab and released in 1990 as a package management application allowing the activation of consistent sets of built software of specified versions and variants (compiler, C++ language standard and build configuration options) in a UNIX shell environment. Exactly one version and variant of any given product can be active in an environment at any time. In more recent incarnations UPS packages can be trivially relocatable. UPS does not prescribe the build system for each product but relies on auxiliary (*table* and *version*) files to instruct it on the relationships between packages and what constitutes activating or deactivating a package (*e.g.* alterations to environment variables such as `PATH`, `LD_LIBRARY_PATH` or `PYTHONPATH`). UPS does not have a build description language so its use is supplemented by the *ad hoc* script package ssibuildshims.

The build instructions for external products and associated auxiliary files such as patches and *table* files are stored in their own source control repository. Products developed in-house such as the *art* suite itself, and experimental software are built using CMake[8] in conjunction with a Fermilab-developed macro package (cetbuildtools[9]) to facilitate building, testing and producing UPS packages. The high level macros provide high level directives for common operations like comparing test output against a reference, or generating and building auxiliary data for the ROOT application[10, 11]. They also handle coordination across multiple aspects of CMakefunctionality for common operations such as ensuring that built libraries and executables are available for use in dependent packages, or encoding the setup of the required dependencies into a generated *table* file. A further system—Multi-Repository Build, or MRB—allows the simultaneous development, build and test of multiple cetbuildtools-based packages against a local or central install of packages upon which they depend. Additional scripts and utilities assist in the tagging and production of releases and the upload of binary packages to the distribution site.

The *art* suite is supported on the current and immediately-previous Scientific Linux and related distributions, the current and immediately-previous macOS, and (on a best effort basis) the current Ubuntu and related distributions. In addition, the art suite has occasionally been used on HPC systems, usually with the aid of containerization software such as Docker, Shifter or Singularity, but the required effort level for each release relative to off-the-shelf platforms is quite high given the particular characteristics of each HPC machine. We also support multiple GCC compilers on Linux and multiple Clang compilers on both Linux and macOS.

The recent ramping-up of efforts across HEP to utilize HPC systems—not a natural fit due to the embarrassingly-parallel character of most event-based HEP algorithms—points towards the mainstreaming and streamlining of support for those platforms. In addition, while client experiments are keen to continue running the software on macOS it is necessary currently to disable the SIP security feature. SIP is fundamentally incompatible with the environment-based method used by UPS to provide binary relocatability due to SIP's enforced sanitization of environment variables such as `DYLD_LIBRARY_PATH`. It is expected that the ability to disable SIP will be removed in a future incarnation of macOS. Client experiments have also long expressed the desire for flexibility in the versions of some software in a release (*e.g.* Geant4) for evaluation purposes, an attribute the current system does not possess. With the age of UPS and the effort involved in supporting it as platforms and packaging and deployment requirements evolve, it is time to look for alternative ways of satisfying those requirements.

The packaging group of the HEP Software Foundation (HSF)[12, 13] has been addressing issues like this common to HEP as a whole over the last few years. It is intended and hoped that a solution resulting from the work described here would have applicability and eventual use across the wider HEP community.

3 Spack as the Basis for a New System

> Spack is a package management tool designed to support multiple versions and configurations of software on a wide variety of platforms and environments.[14]

Originating from LLNL and written in Python with an open source license, Spack was originally conceived to satisfy the requirements of HPC system administrators for a package management tool. It features almost 2900 built-in package recipes ("specs") as of October 2018 with more added almost daily, and the ability to use external repositories of specs in a configured hierarchy. Spack has a "build language" of provided Python functions and objects to allow users to write their own specs. Specs can have variants which may be boolean (`+python`) or value-based (`nmxhep=4000`, `cxxstd=17`) and these variants may be required (or not) when specifying dependencies. Multiple versions and/or variants of a given package may be installed contemporaneously. Different compilers may be used, which become dependencies that must be satisfied consistently within a given dependency tree. Working in conjunction with a module management system (Environment Modules, Lmod, and dotkit are all supported), packages can be activated for use within a UNIX environment. In addition, the Spack community is accessible and open to contributions, not only of new specs but also improvements to specs, and bugfixes and new features for core code.

4 Evolving a Product for a New Community: Extending Spack to Meet New Requirements

As conceived and implemented currently, Spack, like UPS before it, was not intended to satisfy the full set of requirements we have for our whole system. Some of those requirements, such as the need to support individual and simultaneous multiple product development, can be met by external tools as described in section 5. Others are more intimately related to package building and are more appropriately satisfied by enhancements to Spack itself. Buildcache is an enhancement to Spack to enable the production of relocatable binary packages and their upload to (and installation from) a local or remote cache. A proposal and initial implementation of a new Spack command, `spack chain`, is currently being fleshed out by the community and will provide the ability to "layer" Spack installations and take advantage of centrally-installed packages while augmenting the package set with those installed.

Spack's treatment of compiler dependencies is currently quite rigid: every package depends upon the compiler active when the package was built. Very often, for binary-only, C-only or data-only packages, we do not care that a dependency might have been "built" against a different compiler than a package requiring it. Complex variant propagation (making sure all relevant packages are built with the same C++ standard, for example) is also currently quite cumbersome. Additionally, the reliance of the package ID hash calculation mechanism on the current spec rather than being baked into a given package build presents challenges for situations involving multiple releases of evolved product sets installed at the same time, and some work and investigation remains in order to resolve them.

Other opportunities for enhancement have arisen, and will continue to crop up as development continues: pull requests are welcomed, with a friendly and reasonable review, test and merge system in place to incorporate them into the Spack code base. For cases where enhancements are not yet ready for prime time, a fork from the GitHub repository serves to allow progress on the wider goals while keeping up with developments in the Spack codebase. Enhancements which have been accepted into the Spack codebase bring the base application closer to meeting our requirements, and will evolve as necessary with other contributions and enhancements to Spack more naturally and with less effort than maintaining a separate fork would require.

5 Facilitating HEP Product Development: cetmodules and SpackDev

We have created two tools, cetmodules and SpackDev[15] to replace, respectively, cetbuildtools and MRB. cetmodules is, like its predecessor, a CMake macro system, but with its reliance on the presence of UPS and the environment it sets up for each package removed in favor of CMake idioms such as `find_package()`. SpackDev is an external Python application that interrogates Spack for the information necessary to generate a working area with checked-out sources for one or more products to be developed together. It ensures that lower level dependencies are built in the correct way, and that any intermediate products required for consistency are checked out in addition to the ones specified explicitly.

SpackDev's principal command is `init`, which sets up a "SpackDev area" containing: a `srcs` directory where the products to develop are checked out and the generated top-level `CMakeLists.txt` file is placed; the `build` and `install` trees; and `spackdev-aux`, where support files and wrappers are placed for use by the build and by SpackDev itself.

The `init` command invokes Spack in order to obtain information about dependencies, and the products to be developed. Spack reads relevant specs and information about already-installed packages in order to "concretize" the dependency information into a fully-specified set of packages to be built. SpackDev ascertains whether any intermediate products must be checked out in addition to those specified in order to maintain consistency during code development, checks out all the specified products and then ensures that Spack correctly builds and installs the correct versions and variants of all dependencies. SpackDev additionally creates a top-level `CMakeLists.txt` file to coordinate the building of the checked-out products after using Spack to extract the build arguments from the specs, and ensures that building of each package can take place in the correct environment.

6 A Milestone: the Minimum Viable Product

In the interests of producing a usable product that could be evaluated by interested clients, a Minimum Viable Product (MVP) was envisaged. Completed and released in August of 2018, the MVP is a source-based build of all of the *art* suite's dependencies from Spack specs for one platform (Scientific Linux 7), compiler (GCC 7.3), and C++ standard (C++17) for one minimally-modified release of *art*, and an implementation of SpackDev and cetmodules capable of allowing the simultaneous development of all ten of the *art* suite's products.

The MVP bootstrap script will create two Spack installations: one for the compiler and other utilities such as git and another for the dependencies. The MVP treats products from the former installation as "system"-installed commands via Spack's `packages.yaml` file. The latter is configured to satisfy dependencies as far as possible from the system via the same method when they can be reasonably be expected to be installed. Specs for the *art* suite are checked out from an external repository, as are support files such as a template for `packages.yaml` and a required file describing the entire product dependency tree in the form of the arguments to a single `spack install` command that would install everything. After the tools are installed, the user sources a generated setup script and is ready to use SpackDev to create a development area, compile necessary dependencies and start code development with (as a demonstration) the *art* suite.

7 Critical evaluation of the MVP

In the implementation of the MVP, every installation is individual, self-contained, and built from source, leading to a disk-space- and time-consuming `init` phase. SpackDev's use of

Spack is external rather than internal, meaning that several Spack commands are executed—each of which has its own concretization step, which could be time-consuming individually and in aggregate.

The necessity of specifying fully the dependency tree as a `spack install` command presents a source of errors: ideally this would be generated from a more simple input file by a utility which would be able to keep version and variant specifications consistent and identify missing information early.

The concretization time is heavily dependent on the complexity of the dependency graph. The *art* suite's dependency graph has in excess of 2000 edges, and a single concretization operation can take on the order of a minute. The dependency graph for LArSoft[16, 17]—a multi-package toolkit for simulation and reconstruction of HEP events in liquid argon Time Projection Chambers (TPCs)—is significantly more complicated, taking possibly several times longer to concretize, and the results are not cached in any way.

The SpackDev system can only accommodate the development of CMake-based products currently due to the way the build arguments must be extracted from Spack. Also, due to technical details of the way the products are specified to CMake, a top-level parallel build cannot include tests, as the coordinated level of parallelism is lost and tests are executed serially. Due to this, and the difficulty of determining a suitable "best common environment for development," Code–build–test-debug cycles are best implemented using the `spackdev env` command, which puts one's shell into the appropriate environment for one of the products to be developed. From here, the tests for a particular package can be executed and debugged as a developer might expect, dropping back to a global build for integration verification.

8 Remaining Issues and Future Work

The MVP was deliberately limited in scope, leaving several known issues unaddressed in the interests of producing something functional enough to be evaluated meaningfully in the context of our overall requirements. Even so, several issues were identified by the MVP. The problem of multiple croncretization may be conceivably addressed by taking advantage of a not-yet-merged third-party contribution to Spack and reformulating SpackDev as an extension to Spack. Invoked as a Spack command (*e.g.* `spack dev init`) utilizing Spack's functionality as Python modules, the concretization step could be done only once. Currently-cumbersome and *ad hoc* steps such as the extraction environment and build arguments could be regularized and enhanced when done utilizing the SpackAPI rather than the command line interface. The global versus per-product build dichotomy would stand some investigation and possible improvement; regardless, adding parallel test executions into the global build is a desirable goal if technically feasible. Improving the speed and flexibility of the concretization step is certainly desirable, and long term efforts to that effect by core Spack developers are expected to bear fruit in the next few months.

Use of the buildcache facility of Spack will be integrated into the system in the current months, and as the feature matures, layered Spack installations using `spack chain`. Release manamgement utilities will be developed in order to facilitate consistent version, variant and (where appropriate) compiler flag specification across a release in for consistent SpackDev development areas.

In Spack proper, we will be taking advantage of the upcoming improvements to concretization, and investigating how to enhance Spack to allow non-compiler-dependent (*e.g.* data-only and binary-only) products. "Umbrella" (no-source) products are also desirable to facilitate simple setup of experiment analysis environments. Also, a significant task will be investigating how to handle the availability of multiple releases, and the reuse of already-built packages between same in the face of hash changes due to spec evolution.

9 Acknowledgments

We gratefully acknowledge the Spack community in general, and its principal authors in particular, for their help.

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

References

[1] C. Green, J. Kowalkowski, M. Paterno, M. Fischler, L. Garren, Q. Lu, *The Art Framework*, in *Proceedings, 19th International Conference on Computing in High Energy and Nuclear Physics (CHEP 2012): New York, USA, May 21–25, 2012* (2012), Vol. 396, p. 022020

[2] *The art event processing framework*, http://art.fnal.gov/

[3] *Who uses art?*, https://web.archive.org/web/20181030144907/http://art.fnal.gov/who-uses-art/ (2018)

[4] T. Gamblin, M. LeGendre, M.R. Collette, G.L. Lee, A. Moody, B.R. de Supinski, S. Futral, *The Spack Package Manager: Bringing Order to HPC Software Chaos*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (ACM, New York, NY, USA, 2015), SC '15, pp. 40:1–40:12, ISBN 978-1-4503-3723-6, http://doi.acm.org/10.1145/2807591.2807623

[5] *Spack on github*, https://github.com/spack/spack

[6] M. Votava, W. Bliss, S. Cutts-Bone, C. Debaun, F. Donno-Raffaelli, R. Herber, K. Leininger, B. Lindgren, J. Nicholls, G. Oleynik et al., *UPS UNIX Product Support*, in *Seventh Conference Real Time '91 on Computer Applications in Nuclear, Particle and Plasma Physics Conference Record* (1991), pp. 156–159

[7] *The ups redmine project*, https://cdcvs.fnal.gov/redmine/projects/ups

[8] Kitware, Inc., *CMake*, http://cmake.org (2012), http://cmake.org/

[9] *The cetbuildtools project*, https://cdcvs.fnal.gov/redmine/projects/cetbuildtools

[10] R. Brun, F. Rademakers, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **389**, 81 (1997), new Computing Techniques in Physics Research V

[11] *ROOT*, https://root.cern.ch

[12] https://hepsoftwarefoundation.org/, *The hep software foundation*,

[13] *The hep software foundation, packaging working group*, https://hepsoftwarefoundation.org/workinggroups/packaging.html

[14] *Spack documentation*, https://web.archive.org/web/20180717162302/https://spack.readthedocs.io/en/latest/ (2018)

[15] *SpackDev on github*, https://github.com/amundson/spackdev, https://github.com/chissg/spackdev

[16] E. Snider, G. Petrillo, *LArSoft: toolkit for simulation, reconstruction and analysis of liquid argon TPC neutrino detectors*, in *Proceedings, 22nd International Conference on Computing in High Energy and Nuclear Physics, (CHEP 2016), San Francisco, USA, Oct 10–14, 2016* (2017), Vol. 898, p. 042057, http://stacks.iop.org/1742-6596/898/i=4/a=042057

[17] *The LArSoft project*, https://larsoft.org/